

Assignment 11.2

Task Description -1 – (Stack Using AI Guidance)

- **Task:** With the help of AI, design and implement a Stack data structure supporting basic stack operations.

Expected Output:

- A Python Stack class supporting push, pop, peek, and empty-check operations with proper documentation.

The screenshot shows a VS Code editor with a Python file named `task1.py` containing a `Stack` class implementation. The class uses a Python list internally to store elements. The terminal output shows the following sequence of operations:

```
Pushing elements: 10, 20, 30
Current stack: Stack([10, 20, 30])

Top element (peek): 30

Popping elements:
Popped: 30
Popped: 20
Current stack: Stack([10])

Is stack empty? False
Stack size: 1
```

The right sidebar shows a chat window titled "DESIGNING A STACK DATA STRUCTURE IN PYTHON" with a tip to try Plan mode and a list of requirements for the Stack class.

Implemented a Stack class backed by a Python list.

Methods:

- `push(item)`: adds an element to the top.
- `pop()`: removes and returns the top element, raises `IndexError` if empty.
- `peek()`: returns (without removing) the top element, raises `IndexError` if empty.
- `is_empty()`: returns `True` if there are no elements.

Task Description -2 – (Queue Design)

- Task: Use AI assistance to create a Queue data structure

following FIFO principles

Expected Output:

- A complete Queue implementation including enqueue, dequeue, front element access, and size calculation

The screenshot shows a VS Code editor with a Python file named `task2.py` open. The code implements a Queue class using `collections.deque` for efficient operations. The chat window on the right provides instructions and feedback on the implementation.

```
1 """Queue implementation in Python following FIFO (First-In, First-Out) principle.
2
3 This module defines a Queue class with the following operations:
4 - enqueue(item)
5 - dequeue()
6 - front()
7 - size()
8
9 It also includes example usage at the bottom.
10 """
11
12 from collections import deque
13
14
15 class Queue:
16     """A simple FIFO (First-In, First-Out) queue data structure.
17
18     Internally this uses collections.deque for efficient O(1) operations
19     at both ends.
20     """
21
22     def __init__(self):
23         """Initialize an empty queue.
24
25         """
```

Terminal output:

```
C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task2.py"
Is queue empty? True

Enqueuing elements: 'A', 'B', 'C'
Current queue: Queue(['A', 'B', 'C'])

Front element: A

Dequeuing elements (FIFO order):
Dequeued: A
Dequeued: B
Current queue: Queue(['C'])

Queue size: 1
```

Chat window content:

DESIGNING A STACK DATA STRUCTURE IN PYTHON

Act as a data structures instructor.
Create a Queue data structure in Python following FIFO principles.
Requirements:

- Create a Queue class, implement enqueue(), dequeue(), front(), and size() methods. Add proper comments, provide example usage with output, explain how FIFO works in this implementation. Mention time complexity.

Click to Edit task2.py

✓ Implementing queue in task2.py

I'll open your task2.py and then implement a commented Queue class with example usage and time complexity notes.

✓ Reviewed task2.py

Now I'll write the Queue implementation, comments, explanation, and example usage into task2.py.

✓ Updated task2.py

Here's your Queue implementation and how it satisfies each requirement:

- Implemented a `Queue` class in `task2.py` using `collections.deque` for efficient operations.
- Methods:
 - `enqueue(item)`: adds an element to the rear of the queue (FIFO).

- Implemented a Queue class using `collections.deque` for efficient operations.
- Methods:
 - `enqueue(item)`: adds an element to the rear of the queue (FIFO).
 - `dequeue()`: removes and returns the front element; raises `IndexError` if empty.
 - `front()`: returns (without removing) the front element; raises `IndexError` if empty.
 - `size()`: returns the current number of elements.
 - `is_empty()`: helper to check if the queue has no elements.

Task Description -3 – (Singly Linked List Construction)

- **Task:** Utilize AI to build a singly linked list supporting insertion and traversal.

Expected Output:

- **Correctly functioning linked list with node creation, insertion logic, and display functionality.**

```
"""Simple implementation of a Singly Linked List in Python.

This file defines:
- Node class: Represents a single node in the list
- LinkedList class: Manages the list of nodes

Features implemented:
- Insertion at the end of the list
- Traversal / display of all elements

It also includes example usage at the bottom.
"""

class Node:
    """A single node of a singly linked list.

    Each node stores:
    - data: The value we want to keep
    - next: A reference (link) to the next node in the list, or None

    """
    def __init__(self, data):
        # Store the actual data value in this node
        self.data = data
        # 'next' will point to the next node in the list.
        # For a new node that is not yet connected, it is None.
        self.next = None

class LinkedList:
    """Singly Linked List Implementation.

    Internally, the list is represented by a chain of Node objects.
    The LinkedList keeps a reference to the first node, called 'head'.
    Each node points to the next one, forming a chain.

    head -> [data | next] -> [data | next] -> ... -> None

    """
    def __init__(self):
        # 'head' points to the first node in the list.
        # For an empty list, head is None.
        self.head = None

    def insert_at_end(self, data):
        """Insert a new node with the given data at the end of the list.

        Steps:
        1. Create a new node.
        2. If the list is empty (head is None), make this node the head.
        3. Otherwise, traverse to the last node and link its 'next' to the new node.

        """
        new_node = Node(data)

        # Case 1: The list is empty; the new node becomes the first node (head).
        if self.head is None:
            self.head = new_node
            return

        # Case 2: The list is not empty; find the last node.
        current = self.head
```

(c) Microsoft Corporation. All rights reserved.

```
C:\Users\akhil\OneDrive\Documents\Devops>python -u "C:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task3.py"
Inserting elements at the end: 10, 20, 30

Linked list (display):
10 -> 20 -> 30 -> None

Linked list as Python list (traverse):
[10, 20, 30]

C:\Users\akhil\OneDrive\Documents\Devops>
```

- Added Node and LinkedList classes.
- Node holds data and a next reference that points to the next node (or None for the last node).
- LinkedList:
 - Maintains head (first node).
 - insert_at_end(data): creates a new node and links it at the end by walking from head to the last node and updating its next.
 - traverse(): walks from head via next, collects data values into a Python list, and returns it.
 - display(): prints the list in the form 10 -> 20 -> 30 -> None.

Task Description -4 – (Binary Search Tree Operations)

- **Task: Implement a Binary Search Tree with AI support focusing on insertion and traversal.**

Expected Output:

- **BST program with correct node insertion and in-order traversal output.**

```
1  """Binary Search Tree (BST) implementation in Python.
2
3  This file defines:
4  - Node class: represents a single node in the BST
5  - BinarySearchTree class: supports Insertion and in-order traversal
6
7  Features implemented:
8  - Insert a value into the BST
9  - In-order traversal (Left, Root, Right)
10
11 Example usage is provided at the bottom.
12 """
13
14
15 class Node:
16     """A single node in a Binary Search Tree.
17
18     Each node stores:
19     - data: the value stored at this node
20     - left: reference to the left child (values < data)
21     - right: reference to the right child (values > data)
22     """
23
24     def __init__(self, data):
25         self.data = data
26         self.left = None
27         self.right = None
28
29
30 class BinarySearchTree:
31     """Binary Search Tree (BST) with recursive insertion and in-order traversal."""
32
33     def __init__(self):
34         # Root of the BST. Initially, the tree is empty so root is None.
35         self.root = None
36
37     def insert(self, value):
38         """Insert a value into the BST.
```

Chat window content:

DESIGNING A STACK DATA STRUCTURE IN PYTHON

Act as a data structures professor.
Help me implement a Binary Search Tree in Python.
Requirements:

- Create a Node class
- Implement BST insertion method
- Implement In-order traversal
- Add proper comments explaining recursion
- Provide example with sample values
- Show expected traversal output
- Explain why in-order traversal gives sorted output

task4.py

✓ Implementing BST in Python

I'll open task4.py (currently empty) and then add a Binary Search Tree implementation with insertion, in-order traversal, comments, and example usage.

✓ Reviewed task4.py

Now I'll write the BST Node and tree classes, plus example and explanations, into task4.py.

✓ Updated task4.py

Here's what I've implemented for your Binary Search Tree:

- Added `Node` and `BinarySearchTree` classes in `AI-ASSISTED_CODING/task4.py`.
- `Node` holds `data`, `left`, and `right` references.
- `BinarySearchTree`...

task5.py + task4.py

Describe what to build next

Python 3.13 (64-bit) (Go Live) Windsurf (L...) Prettier

Added Node and BinarySearchTree classes in AI-ASSISTED_CODING/task4.py.

Node holds data, left, and right references.

BinarySearchTree:

insert(value): public insert method using _insert_recursive(node, value):

Base case: if node is None, create and return a new Node.

Recursive case: go left if value < node.data, right if value > node.data, then return node to maintain links.

inorder_traversal(): returns a list of values; uses _inorder_recursive(node, result):

Base case: node is None → return.

Recursive case: traverse left, visit node (append data), traverse right.

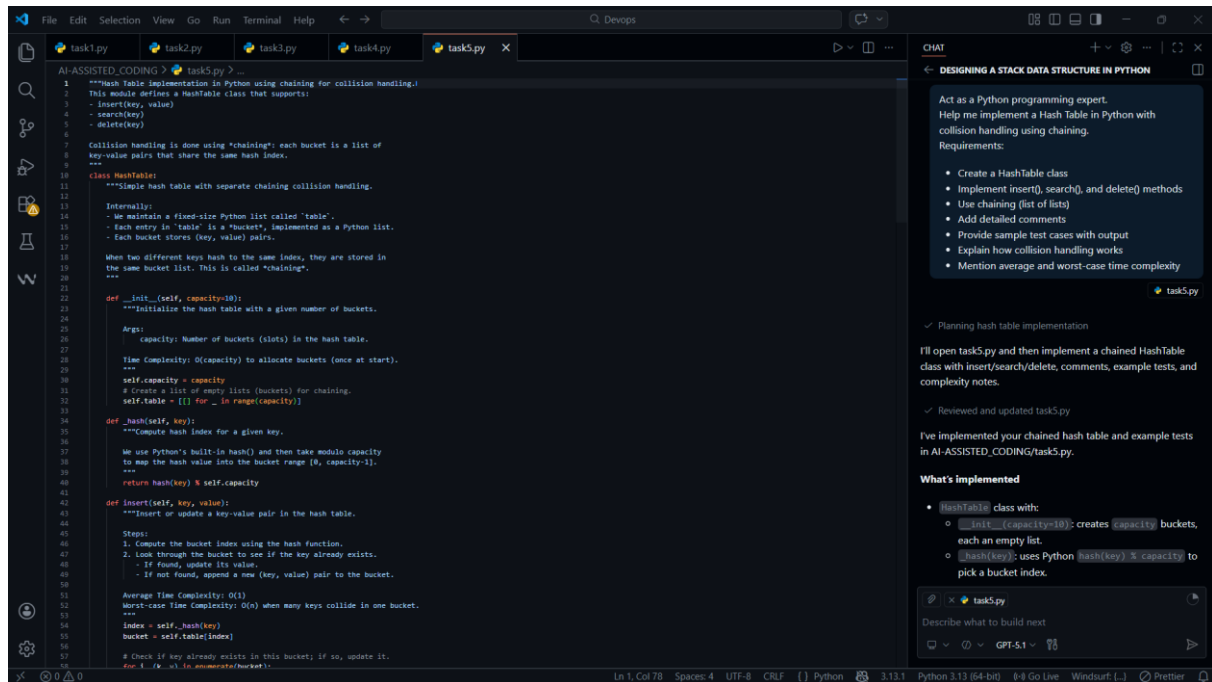
Comments in both recursive helpers explain base/recursive cases clearly.

Task Description -5 – (Hash Table Implementation)

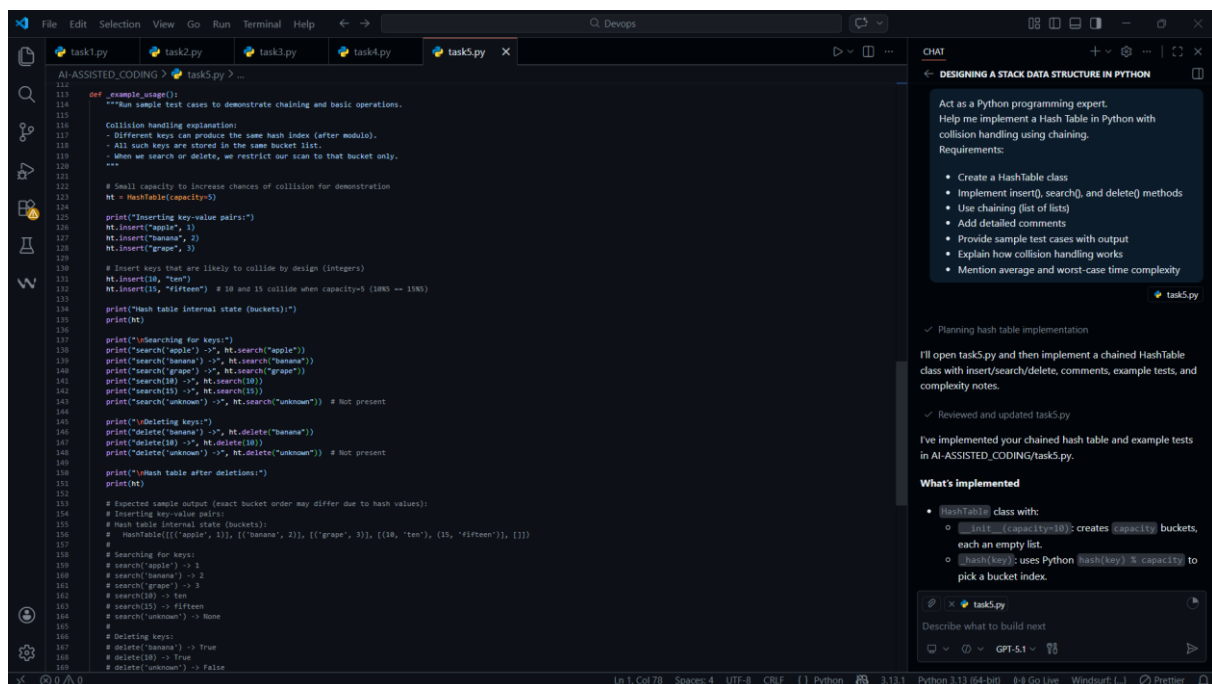
• Task: Create a hash table using AI with collision handling

Expected Output:

• Hash table supporting insert, search, and delete using chaining or open



```
1  """Hash Table Implementation in Python using chaining for collision handling.
2  This module defines a HashTable class that supports:
3  - insert(key, value)
4  - search(key)
5  - delete(key)
6
7  Collision handling is done using "chaining": each bucket is a list of
8  key-value pairs that share the same hash index.
9
10 """
11
12 class HashTable:
13     """Simple hash table with separate chaining collision handling.
14
15     Internally:
16     - We maintain a fixed-size Python list called 'table'.
17     - Each entry in 'table' is a "bucket", implemented as a Python list.
18     - Each bucket stores (key, value) pairs.
19
20     When two different keys hash to the same index, they are stored in
21     the same bucket list. This is called "chaining".
22     """
23
24     def __init__(self, capacity=10):
25         """Initialize the hash table with a given number of buckets.
26
27         Args:
28             capacity: Number of buckets (slots) in the hash table.
29
30         Time Complexity: O(capacity) to allocate buckets (once at start).
31         """
32         self.capacity = capacity
33         # Create a list of empty lists (buckets) for chaining.
34         self.table = [[] for _ in range(capacity)]
35
36     def _hash(self, key):
37         """Compute hash index for a given key.
38
39         We use Python's built-in hash() and then take modulo capacity
40         to map the hash value into the bucket range [0, capacity-1].
41         """
42         return hash(key) % self.capacity
43
44     def insert(self, key, value):
45         """Insert or update a key-value pair in the hash table.
46
47         Steps:
48         1. Compute the bucket index using the hash function.
49         2. Look through the bucket to see if the key already exists.
50            - If found, update its value.
51            - If not found, append a new (key, value) pair to the bucket.
52
53         Average Time Complexity: O(1)
54         Worst-case Time Complexity: O(n) when many keys collide in one bucket.
55         """
56         index = self._hash(key)
57         bucket = self.table[index]
58
59         # Check if key already exists in this bucket; if so, update it.
60         for i, (k, v) in enumerate(bucket):
61             if k == key:
62                 bucket[i] = (key, value)
63                 return
64         # Key not found, append new pair to the bucket.
65         bucket.append((key, value))
66
67     def search(self, key):
68         """Search for a key in the hash table and return its value.
69
70         Steps:
71         1. Compute the bucket index using the hash function.
72         2. Look through the bucket to find the key.
73            - If found, return the value.
74            - If not found, return None.
75
76         Average Time Complexity: O(1)
77         Worst-case Time Complexity: O(n) when many keys collide in one bucket.
78         """
79         index = self._hash(key)
80         bucket = self.table[index]
81
82         for k, v in bucket:
83             if k == key:
84                 return v
85         return None
86
87     def delete(self, key):
88         """Delete a key-value pair from the hash table.
89
90         Steps:
91         1. Compute the bucket index using the hash function.
92         2. Look through the bucket to find the key.
93            - If found, remove it from the bucket.
94            - If not found, return False.
95
96         Average Time Complexity: O(1)
97         Worst-case Time Complexity: O(n) when many keys collide in one bucket.
98         """
99         index = self._hash(key)
100        bucket = self.table[index]
101
102        for i, (k, v) in enumerate(bucket):
103            if k == key:
104                del bucket[i]
105                return True
106        return False
```



```
111 def _example_usage():
112     """Run sample test cases to demonstrate chaining and basic operations.
113
114     Collision handling explanation:
115     - Different keys can produce the same hash index (after modulo).
116     - All such keys are stored in the same bucket list.
117     - When we search or delete, we restrict our scan to that bucket only.
118     """
119
120     # Small capacity to increase chances of collision for demonstration
121     ht = HashTable(capacity=5)
122
123     print("Inserting key-value pairs:")
124     ht.insert("apple", 1)
125     ht.insert("banana", 2)
126     ht.insert("grape", 3)
127
128     # Insert keys that are likely to collide by design (integers)
129     ht.insert(10, "ten")
130     ht.insert(15, "fifteen") # 10 and 15 collide when capacity=5 (10%5 == 15%5)
131
132     print("Hash table internal state (buckets):")
133     print(ht.table)
134
135     print("\nSearching for keys:")
136     print(search("apple") -> ", ht.search("apple"))
137     print(search("banana") -> ", ht.search("banana"))
138     print(search("grape") -> ", ht.search("grape"))
139     print(search(10) -> ", ht.search(10))
140     print(search(15) -> ", ht.search(15))
141     print(search("unknown") -> ", ht.search("unknown")) # Not present
142
143     print("\nDeleting keys:")
144     print(delete("banana") -> ", ht.delete("banana"))
145     print(delete(10) -> ", ht.delete(10))
146     print(delete("unknown") -> ", ht.delete("unknown")) # Not present
147
148     print("Hash table after deletions:")
149     print(ht.table)
150
151     # Expected sample output (exact bucket order may differ due to hash values):
152     # Inserting key-value pairs:
153     # Hash table internal state (buckets):
154     # HashTable([['apple', 1], ['banana', 2], ['grape', 3], [10, 'ten'], [15, 'fifteen']], [])
155
156     # Searching for keys:
157     # search("apple") -> 1
158     # search("banana") -> 2
159     # search("grape") -> 3
160     # search(10) -> ten
161     # search(15) -> fifteen
162     # search("unknown") -> None
163
164     # Deleting keys:
165     # delete("banana") -> True
166     # delete(10) -> True
167     # delete("unknown") -> False
```

```

Inserting values into BST: [50, 30, 70, 20, 40, 60, 80]

In-order traversal result:
[20, 30, 40, 50, 60, 70, 80]

Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]

In-order traversal result:
[20, 30, 40, 50, 60, 70, 80]

Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]
Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]

C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task5.py"
Inserting key-value pairs:
Hash table internal state (buckets):
HashTable([(['apple', 1), (10, 'ten'), (15, 'fifteen')], [], [], [(['grape', 3)], [(['banana', 2)])])

Searching for keys:
search('apple') -> 1
search('banana') -> 2
search('grape') -> 3
search(10) -> ten
search(15) -> fifteen
search('unknown') -> None

Deleting keys:
delete('banana') -> True
delete(10) -> True
delete('unknown') -> False

```

- HashTable class with:
 - `__init__(capacity=10)`: creates capacity buckets, each an empty list.
 - `__hash(key)`: uses Python `hash(key) % capacity` to pick a bucket index.
 - `insert(key, value)`: updates existing key or appends (key, value) into the bucket.
 - `search(key)`: scans the bucket for key, returns the value or None.
 - `delete(key)`: removes (key, value) from the bucket, returns True/False.
- Chaining:
 - `self.table` is a list of lists (buckets).
 - Each bucket stores multiple (key, value) pairs that share the same index → this is collision handling by chaining.